Mushroom: a program for the Automated Verification of an SCM Protocol Specification

G. M. Lundy and B. Bulbul
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

Abstract

Systems of Communicating Machines (SCM) is a formal model for the specification, verification and testing of communication protocols, which has been used to specify and verify several well-known protocols. In this paper we discuss a program, called *mushroom*, which automates the analysis using the SCM model. The program generates either the system state analysis or the full reachability analysis. The automation of this model is expected to greatly facilitate the use of the model for protocol design and analysis. The program has been used to verify some theoretical results, serving to confirm these earlier results.

1 Introduction

Systems of communicating machines [LuAk - LuMi] is a formally defined model intended for the specification, analysis and testing of communication protocols. This model uses a combination of finite state machines and variables, which may be local to a single machine or shared by two or more machines; so it falls into the class of models known as "extended finite state machines." Each station or computer in the network is modeled by a finite state machine and associated *local variables*. Accompanying each transition in the machine is an *action*, which may alter the variable values. Machines communicate with other machines through *shared variables*. *Enabling predicates* determine when a transition may be taken, and *actions* are taken when the transition is executed, where the action assigns new values to some subset of the variables.

Analysis for protocols specified with this model can be carried out using a method called *system state analysis*. This analysis is similar to global reachability analysis, which generates all states reachable from the initial state, but generates a subset of all reachable states, which is generally a fraction of the size.

In this paper we describe a program, called **mushroom**, which has been written to automate protocol analysis, which generates either the system state analysis (**smart mushroom**), or the full global analysis (**big mushroom**) for a protocol specified formally as a *system of communicating machines*. The program is also able to accept protocols specified as *communicating finite state machines* and generating the global reachability analysis. The name "mushroom" was chosen as a symbol of something which starts out relatively small (specification) and gets much bigger quickly (analysis).

Much research has been done in the past 15 years on the formal modeling of protocols. In addition to the early CFSM model, three different models have been chosen for standardization; these are Estelle, LOTOS, and SDL [Bel,Bri,Bud]. A number of tools have also been written for the design and verification of protocols; [Agg]. is one example. Undoubtedly these models and tools are quite useful. However our conviction that *systems of communicating machines* is rich enough to express almost any protocol or algorithm, simple enough for verification, and that a procedure for conformance testing has been defined [MiLu] has led to its continued development. With this paper, we present a software tool that we hope will be a major step towards increasing its usefulness to the protocol community.

In the next section the model *systems of com-*

U.S. Government Work Not Protected by U.S. Copyright

municating machines protocol is described. The program (mushroom) is described in the third section. In section 4, some examples are given, and concluding remarks follow in section 5.

2 Systems of Communicating Machines

A detailed description appears in references [LuAk, LuMi].

A *system of communicating machines* is an ordered pair $C = (M, V)$, where M is a finite set of *machines*, and V is a finite set of *shared variables*.

Each machine $m_i \epsilon M$ is defined by a tuple $(S_i, s, L_i, N_i, \tau_i)$, where (1) S_i is a finite set of states; (2) $s \epsilon S_i$ is a designated state called the *initial state* of m_i; (3) L_i is a finite set of *local variables*; (4) N_i is a finite set of names, each of which is associated with a unique pair (p, a), where p is a predicate and a an *action* on the local and shared variables; (5) $\tau_i : S_i \times N_i \rightarrow S_i$ is a transition function, which is a partial function from the states and names of m_i to the states of m_i.

A *system state tuple* is a tuple of all machine states; a *system state* is a system state tuple, plus the outgoing transitions which are enabled; and a *global state* is a system state tuple, plus the values of all variables, both local and shared.

System state analysis is the process of generating the set of all system states reachable from the initial system state. This analysis constructs a graph whose nodes are the reachable system states, and whose arcs indicate the transitions leading from each system state to another. The algorithm is described in detail in [LuMi].

Example. To illustrate the definitions above, a simple example of a protocol specification and analysis is given. We will use the alternating bit protocol, because it is both simple and well known by the protocol community.

The specification consists of the finite state machines, the local and shared variables, shown in Figure 1, and the predicate-action table, shown in Table 1. The initial state of each machine is 0, with the shared variables empty, and the *seq* and *exp* local variables initially 0.

The system state analysis for this protocol generates four system states, as shown in Figure 2. Upon returning to the initial system state, the reader may observe that the values in the local variables *seq* and *exp* are not the same as in the initial state. Thus, this is a different *global* state than the initial global state; however, since the enabled outgoing transitions are the same, by definition, it is the same *system* state.

Initially both machines are in state 0, and the local variables *seq* and *exp* are set to 0. The subscripts are used so that distinct system states having the same tuple (but not the same outgoing transitions) may easily be distinguished.

The reader may easily verify that the analysis is as shown in Fig. 2, and that upon entering the initial state $< 0, 0 >$ for the second time, that the values of *seq* and *exp* are now 1; that is, the initial *global* state has not yet been reached. In order to reach the initial global state (thus completing the global analysis), the analysis must be continued through four more transitions, generating a total of eight global states. The complete global state analysis is shown in [Bul].

Thus, for this protocol we have 4 system states, and 8 global states. For more complex protocols, the difference between these numbers can be much greater. For example, the sliding window protocol generates 11880 global states and 165 system states for a window size of 8.

3 Mushroom: a program for automated protocol analysis

Figure 3 shows the general structure of Mushroom; there are three parts. The first, "Simple Mushroom," gives the analysis for the CFSM model, finite state machines without variables. The remaining two parts accept a specification of a protocol

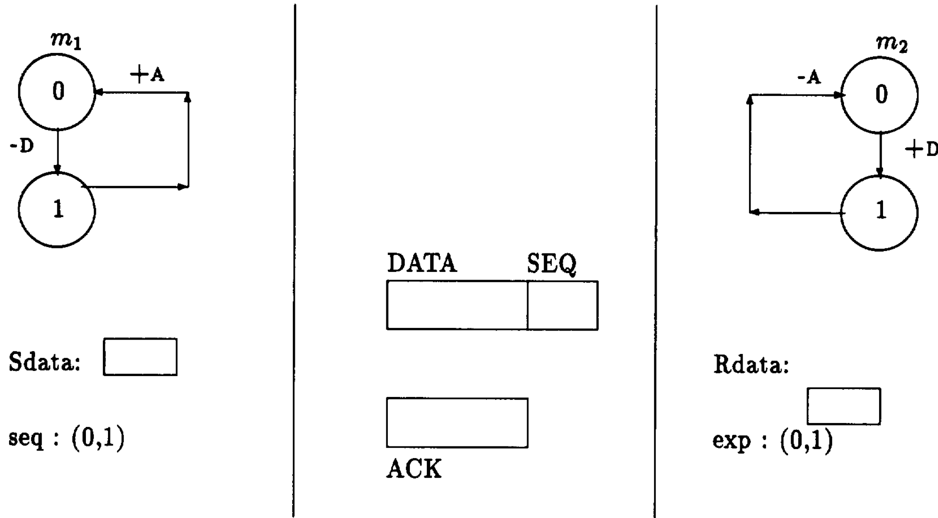

Figure 1 : Specification for the Alternating Bit Protocol

trans	*enabling predicate*	*action*
$-D$	$DATA = \mathcal{E} \wedge$ $SEQ = \mathcal{E}$	$DATA \leftarrow Sdata$ $SEQ \leftarrow seq$ $inc(seq)$
$+A$	$ACK \neq \mathcal{E}$	$ACK \leftarrow \mathcal{E}$
$+D$	$DATA \neq \mathcal{E} \wedge$ $SEQ = exp$	$Rdata \leftarrow DATA$ $DATA, SEQ \leftarrow \mathcal{E}$; $inc(exp)$
$-A$	$DATA = \mathcal{E}$	$ACK \leftarrow exp$; $Rdata \leftarrow \mathcal{E}$

Table 1: P-A Table for Alt-Bit Protocol

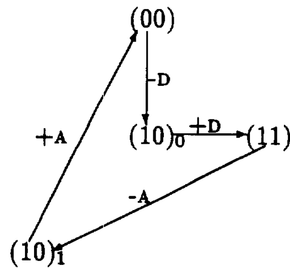

Figure 2 : System State Analysis for the Alt-Bit Protocol

as a *system of communicating machines*. The finite state machine is specified the same as with *simple mushroom*; the variables and tables require additional input. *Big Mushroom* generates the full global analysis, while *smart mushroom* generates the smaller system state analysis.

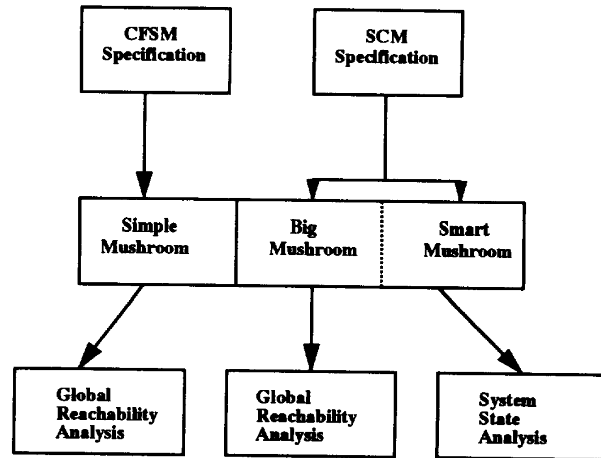

Figure 3: General Structure of Mushroom Program

3.1 Simple Mushroom: a program for CFSM analysis

Input. The CFSM model specification of a protocol consists of only FSM's of the communicating machines. In the program FSM's are represented with a text file. The user enters the directed graphs as a text file using reserved words.

Reachability Analysis. After reading the input file the program starts generating the global reachability graph. Starting with the initial state, the new states are added and linked dynamically. During the graph construction, program also detects the global states with deadlocks and unspecified receptions. The program also finds the maximum message queue size and channel overflows. A maximum channel capacity is introduced for ensuring that the analysis stops eventually.

Output. The program stores the analysis results in a file named by the user during the reachability graph construction. This file contains the specification in a tabular format, reachability graph and the results of the analysis consisting of the number of states generated, number of states analyzed, number of deadlocks, number of unspecified receptions, maximum message queue size and number of channel overflows. Global states with deadlocks and unspecified receptions are also marked in the reachability graph. The output file also lists the unexecuted transitions. A menu is displayed at the end of the analysis. The user can choose from this menu for displaying or printing the results or can continue the program for another analysis.

If the analysis generates more than 2000 states the program gives an interim summary of the analysis after generating 2000 states and asks the user whether to continue. If the user wishes to continue, analysis procedes in steps of 1000 states until the analysis ends or the user terminates the analysis (so long as memory is available).

3.2 Smart and Big Mushrooms

Inputs. Specification of a protocol in the SCM model consists of three parts: the finite state machine (FSM), the local and shared variable definitions, and the predicate-action table. The FSMs are represented in the same manner as described in the previous section. Variable definitions and predicate-action table are entered using Ada formatted packages and procedures. The user fills in templates which are provided by the program, and compiles them together with program.

Construction of the specification in the form of Ada packages and procedures is explained in the following paragraphs.

Variable Definitions. The user defines the protocol variables in an Ada package named "definitions." This package includes the local variables for each machine and the global variables which are considered shared that allow communication between machines. These types are used to define the protocol variables. The template for the definitions package is given in Figure 4. The user simply completes the definitions package by filling in the necessary places in the template.

Predicate-action Table. The predicate-action table is represented by a number of subprograms as separate compilation units. These subprograms are Analyze_Predicates procedures that determine the enabled transitions for each machine and an Action procedure that executes the actions to be taken for the corresponding enabled predicates. There is one Analyze_Predicates procedure for each machine and one Action procedure for the protocol. The template for the Analyze_Predicates procedure is similar in format, and is shown in [Bul].

The user completes the template for each state of the machines. For each machine state there is one "when" statement. "If" statements specify the predicates for possible transitions from the current state. The "Push" statement stores these transitions in the stack. Since more than one transition can be enabled in some states, a stack is used to store all possible transitions.

For each action that must be taken, there is one Action procedure. The template for this procedure is similar in format, and is given in [Bul].

The enabled transitions are passed into this ac-

```
with TEXT_IO;
use TEXT_IO;
package definitions is
  num_of_machines : constant := ████████;
  type scm_transition_type is ( ████████████ );
  type dummy_type is range 1..255;

  type machine1_state_type is
    record
      dummy : dummy_type;
      ████████████
    end record;
.
.
  type machine8_state_type is
    record
      dummy : dummy_type;
      ████████████
    end record;
  type global_variable_type is
    record
      ████████████
    end record;
end definitions;
```


Figure 4: Template for definitions package.

tion procedure through the "in_transition" formal parameter and the necessary changes are made to the local and shared variables by the Action procedure. The user needs to complete this template for the actions to be taken for each transition.

Analysis. After entering the specification, system state analysis or global state analysis can be executed.

During the construction of the reachability graph the program also determines the states with deadlock, and if any transitions are left unexecuted.

Output. The program stores the results of the analysis in a file. This file contains FSM's in a tabular format, system/global reachability graph and results of the analysis consisting of number of states generated, number of states analyzed and number of deadlocks. Unexecuted transitions are also listed at the end of the analysis.

Since each protocol specification has different variables, user can also define the output format for the global state tuples. This is done in a similar manner to the predicate-action and variable definitions representation using an Ada procedure template. The user completes the template with Ada "put" statements for outputting the global states. This file must also compiled with the other units of the program prior to the execution.

As in the Simple Mushroom if the analysis generates more than 2000 states, program gives an interim summary and continues in steps as described in section 3.

At the end of the program user can display/print the results or continue with another sytem/global state analysis selecting the desired options from the menu.

4 Some examples of the use of the program

The SCM specification for this protocol was given in section 2. Inputting this specification into the program is explained in the following subsections.

Input: finite state machines. The FSM description is entered using the reserved words listed in section 3.2 in combination with the transition names and state numbers from the directed graph. The input file for the finite state machines is given below.

```
start
num_of_machines 2
machine 1
state 0
trans snd_data 1
state 1
trans rcv_ack 0
machine 2
state 0
trans rcv_data 1
state 1
trans snd_ack 0
initial_state 0 0
finish
```

Transition names snd_data, rcv_data, rcv_ack and snd_ack corresponds to -D, +D, +A and -A respectively.

Variable definitions. The completed definitions package for the example protocol specification is shown in [Bul].

In the package, empty ($\emptyset$) values for shared variable SEQ and ACK are represented by "-1." The other declarations are easily recognized from the SCM specification in section 2.2.

Predicate-action table. The predicate-action table is entered using the templates which were discusses above. The completed template is given in [Bul].

Output format. The user is also given the flexibility to define the output format for global states. This can be done using a template provided to the user, similar to those shown above.

Results of the analysis. Results of both system state analysis and global state analysis which are obtained for the example protocol using the program Mushroom is shown in Figure 5.

5 Conclusions and Further Research Possibilities

In this paper a program has been described which automates the analysis of protocols specified formally using the model *systems of communicating machines*. The program generates either the *system state analysis* – which is a shortened form of reachability analysis – or global analysis. The program also can accept as input a protocol specified formally as a set of *communicating finite state machines*, and generates the set of reachable states. These statements assume, in all three cases, that the set is finite, and within the limits of the machine storage capacity.

The program has been tested against results of several previous works, and found to confirm their results. For example, in [LuMi], exact formulas were given for a sliding window protocol, for both system state and global state analysis, as a function of the window size. The program generated the identical numbers of system and global states for window sizes from two to ten. (For a window size of 10, there were 286 system states and 31,460 global states). In [Lun], 73391 global states were generated for a CFSM specification; our program generated exactly the same result.

This work brings up some research questions concerning the SCM model which have not yet been completely answered. When is the system state analysis sufficient to prove the protocol properties of interest, and when is it not sufficient? For example, if there are no deadlocks in the system state analysis, can we be sure there are none in the global analysis? There are examples of both cases. It is hoped that this program may help us to solve this question. At any rate, the performance of system state analysis can usually answer some questions, and the program provides the option of a full global analysis – memory and time permitting!

The program is a tool which the authors expect to greatly improve the ease with which protocols can be analyzed in this model. However, the basic problem of the combinatorial state explosion still exists. In our view, protocols generating massive numbers of states must be analyzed by a combination of intelligent logic on the part of the designer, with reachability analysis of some parts of the protocol. Finally, simulation and testing also will play a part.

[Agg] Aggarwal S, Barbara D, Meth K Z, "SPANNER: a tool for the specification, analysis and evaluation of protocols," *IEEE Transactions on Software Engineering*, SE-13, 1218-1237, 1987.

[Bel] Belina, F, Hogrefe and Sarma A, *SDL with Applications from Protocol Specification*, BCS Practitioner Series, Prentice-Hall.

[Bri] Brinksma E, "A Tutorial on LOTOS," *Protocol Specification, Testing and Verification V*, North-Holland, 1985.

[Bul] Bulbul B, "Mushroom: A Program for the Automated Analysis of a Formally Specified Protocol," MS thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, June 1993.

[Bud] Budkowski S, Dembinski P, "The formal specification technique estelle," *Computer Networks and ISDN Systems*, V.14, 1987.

[Hol] Holzmann, G. J., "Coverage Preserving Reduction Strategies for Reachability Analysis," in *Protocol Specification, Testing and Verification XII*, North-Holland, 1992.

[Lun] Lundy G. M., "Modeling and Analysis of Data Link Protocols," TN86-499.1, Telecommunications Research Laboratory, GTE Laboratories, 40 Sylvan Road, Waltham, Mass., January, 1986.

[LuAk] Lundy, G. M., and Akyildiz, I. F., "Specification and analysis of the FDDI MAC Protocol Using Systems of Communicating Machines," *Computer Communications*, V.15 No.5, June 1992.

[LuMi] Lundy, G. M., and Miller, Raymond E., "Specification and Analysis of a Data Transfer Protocol Using Systems of Communicating Machines," *Distributed Computing*, Springer-Verlag, December 1991.

[MiLu] Miller, Raymond E., and Lundy, G. M., "Testing Protocol Implementations Based on a Formal Specification," in *Protocol Test Systems, III*, North-Holland, 1991.

REACHABILITY ANALYSIS of alt_bit_protocol.scm

SPECIFICATION

Machine 1 State Transitions

From	To	Transition
0	1	snd_data
1	0	rcv_ack

Machine 2 State Transitions

From	To	Transition
0	1	rcv_data
1	0	snd_ack

SYSTEM REACHABILITY GRAPH

```
0 [ 0, 0 ] 0    snd-data  1
1 [ 1, 0 ] 0    rcv_data  2
2 [ 1, 1 ] 0    snd_ack   3
3 [ 1, 0 ] 1    rcv_ack   0
```

SUMMARY OF SYSTEM REACHABILITY ANALYSIS(ANALYSIS COMPLETED)

number of states generated : 4
number of states analyzed : 4
number of deadlocks : 0
UNEXECUTED TRANSITIONS : NONE

GLOBAL REACHABILITY GRAPH

[m1 , m2 , seq , Sdata , exp , Rdata , DATA , SEQ , ACK]

```
0 [ 0 , 0 , 0 , D, 0 , E , E , -1 , -1 ] snd_data  1
1 [ 1 , 0 , 1 , D , 0 , E , D , 0 , -1 ] rcv_data  2
2 [ 1 , 1 , 1 , D , 1 , D , E , -1 , -1 ] snd_ack  3
3 [ 1 , 0 , 1 , D , 1 , E , E , -1 , 1 ] rcv_ack  4
4 [ 0 , 0 , 1 , D , 1 , E , E , -1 , -1 ] snd_data  5
5 [ 1 , 0 , 0 , D , 1 , E , D , 1 , -1 ] rcv_data  6
6 [ 1 , 1 , 0 , D , 0 , D , E , -1 , -1 ] snd_ack  7
7 [ 1 , 0 , 0 , D , 0 , E , E , -1 , 0 ] rcv_ack  0
```

SUMMARY OF GLOBAL REACHABILITY ANALYSIS(ANALYSIS COMPLETED)

number of states generated : 8
number of states analyzed : 8
number of deadlocks : 0
UNEXECUTED TRANSITIONS : NONE

Figure 5: Program Output